Capítulo 7

Quicksort

O quicksort (ordenação rápida) é um algoritmo de ordenação cujo tempo de execução do pior caso é $\Theta(n^2)$ sobre um arranjo de entrada de n números. Apesar desse tempo de execução lento no pior caso, o quicksort com freqüência é a melhor opção prática para ordenação, devido a sua notável eficiência na média: seu tempo de execução esperado é $\Theta(n \lg n)$, e os fatores constantes ocultos na notação $\Theta(n \lg n)$ são bastante pequenos. Ele também apresenta a vantagem da ordenação local (ver Capítulo 2) e funciona bem até mesmo em ambientes de memória virtual.

A Seção 7.1 descreve o algoritmo e uma sub-rotina importante usada pelo quicksort para particionamento. Pelo fato do comportamento de quicksort ser complexo, começaremos com uma discussão intuitiva de seu desempenho na Seção 7.2 e adiaremos sua análise precisa até o final do capítulo. A Seção 7.3 apresenta uma versão de quicksort que utiliza a amostragem aleatória. Esse algoritmo tem um bom tempo de execução no caso médio, e nenhuma entrada específica induz seu comportamento do pior caso. O algoritmo aleatório é analisado na Seção 7.4, onde mostraremos que ele é executado no tempo $\Theta(n^2)$ no pior caso e no tempo $O(n \lg n)$ em média.

7.1 Descrição do quicksort

O quicksort, como a ordenação por intercalação, se baseia no paradigma de dividir e conquistar introduzido na Seção 2.3.1. Aqui está o processo de dividir e conquistar em três passos para ordenar um subarranjo típico $A[p \,..\, r]$.

Dividir: O arranjo $A[p \,..\, r]$ é particionado (reorganizado) em dois subarranjos (possivelmente vazios) $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ tais que cada elemento de $A[p \,..\, q-1]$ é menor que ou igual a $A[q]$ que, por sua vez, é igual ou menor a cada elemento de $A[q+1 \,..\, r]$. O índice q é calculado como parte desse procedimento de particionamento.

Conquistar: Os dois subarranjos $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ são ordenados por chamadas recursivas a quicksort.

Combinar: Como os subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los: o arranjo $A[p \,..\, r]$ inteiro agora está ordenado.

O procedimento a seguir implementa o quicksort.

```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q − 1)
4         QUICKSORT(A, q + 1, r)
```

Para ordenar um arranjo A inteiro, a chamada inicial é QUICKSORT(A, 1, *comprimento*[A]).

Particionamento do arranjo

A chave para o algoritmo é o procedimento PARTITION, que reorganiza o subarranjo $A[p .. r]$ localmente.

```
PARTITION(A, p, r)
1 x ← A[r]
2 i ← p − 1
3 for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                trocar A[i] ← A[j]
7 trocar A[i + 1] ↔ A[r]
8 return i + 1
```

A Figura 7.1 mostra a operação de PARTITION sobre um arranjo de 8 elementos. PARTITION sempre seleciona um elemento $x = A[r]$ como um elemento ***pivô*** ao redor do qual será feito o particionamento do subarranjo $A[p .. r]$. À medida que o procedimento é executado, o arranjo é particionado em quatro regiões (possivelmente vazias). No início de cada iteração do loop **for** nas linhas 3 a 6, cada região satisfaz a certas propriedades, que podemos enunciar como um loop invariante:

No início de cada iteração do loop das linhas 3 a 6, para qualquer índice de arranjo k,

1. Se $p \le k \le i$, então $A[k] \le x$.
2. Se $i + 1 \le k \le j - 1$, então $A[k] > x$.
3. Se $k = r$, então $A[k] = x$.

A Figura 7.2 resume essa estrutura. Os índices entre j e $r - 1$ não são cobertos por quaisquer dos três casos, e os valores nessas entradas não têm nenhum relacionamento particular para o pivô x.

Precisamos mostrar que esse loop invariante é verdadeiro antes da primeira iteração, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, $i = p - 1$ e $j = p$. Não há nenhum valor entre p e i, e nenhum valor entre $i + 1$ e $j - 1$; assim, as duas primeiras condições do loop invariante são satisfeitas de forma trivial. A atribuição na linha 1 satisfaz à terceira condição.

Manutenção: Como mostra a Figura 7.3, existem dois casos a considerar, dependendo do resultado do teste na linha 4. A Figura 7.3(a) mostra o que acontece quando $A[j] > x$; a única ação no loop é incrementar j. Depois que j é incrementado, a condição 2 é válida para $A[j - 1]$ e todas as outras entradas permanecem inalteradas.

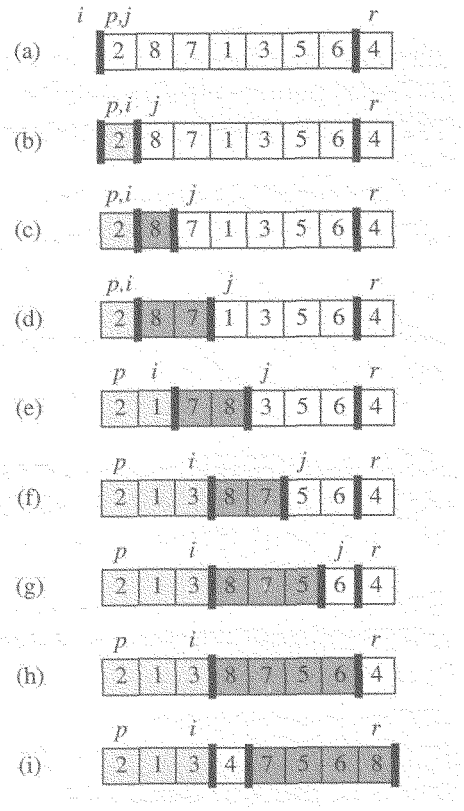

FIGURA 7.1 A operação de PARTITION sobre um exemplo de arranjo. Os elementos do arranjo ligeiramente sombreados estão todos na primeira partição com valores não maiores que x. Elementos fortemente sombreados estão na segunda partição com valores maiores que x. Os elementos não sombreados ainda não foram inseridos em uma das duas primeiras partições, e o elemento final branco é o pivô. (**a**) O arranjo inicial e as configurações de variáveis. Nenhum dos elementos foi inserido em qualquer das duas primeiras partições. (**b**) O valor 2 é "trocado com ele mesmo" e inserido na partição de valores menores. (**c**)–(**d**) Os valores 8 e 7 foram acrescentados à partição de valores maiores. (**e**) Os valores 1 e 8 são permutados, e a partição menor cresce. (**f**) Os valores 3 e 8 são permutados, e a partição menor cresce. (**g**)–(**h**) A partição maior cresce até incluir 5 e 6 e o loop termina. (**i**) Nas linhas 7 e 8, o elemento pivô é permutado de forma a residir entre as duas partições

A Figura 7.3(b) mostra o que acontece quando $A[j] \leq x$; i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Por causa da troca, agora temos que $A[i] \leq x$, e a condição 1 é satisfeita. De modo semelhante, também temos que $A[j-1] > x$, pois o item que foi trocado em $A[j-1]$ é, pelo loop invariante, maior que x.

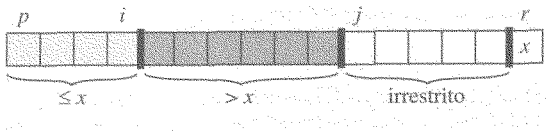

FIGURA 7.2 As quatro regiões mantidas pelo procedimento PARTITION em um subarranjo $A[p \,..\, r]$. Os valores em $A[p \,..\, i]$ são todos menores que ou iguais a x, os valores em $A[i+1 \,..\, j-1]$ são todos maiores que x, e $A[r] = x$. Os valores em $A[j \,..\, r-1]$ podem ser quaisquer valores

Término: No término, $j = r$. Então, toda entrada no arranjo está em um dos três conjuntos descritos pelo invariante, e particionamos os valores no arranjo em três conjuntos: os que são menores que ou iguais a x, os maiores que x, e um conjunto unitário contendo x.

As duas linhas finais de PARTITION movem o elemento pivô para seu lugar no meio do arranjo, permutando-o com o elemento mais à esquerda que é maior que x. A saída de PARTITION agora satisfaz às especificações dadas para a etapa de dividir.

O tempo de execução de PARTITION sobre o subarranjo $A[p \,..\, r]$ é $\Theta(n)$, onde $n = r - p + 1$ (ver Exercício 7.1-3).

Exercícios

7.1-1

Usando a Figura 7. 1 como modelo, ilustre a operação de PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1-2

Que valor de q PARTITION retorna quando todos os elementos no arranjo $A[p \,..\, r]$ têm o mesmo valor? Modifique PARTITION de forma que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p \,..\, r]$ têm o mesmo valor.

7.1-3

Forneça um breve argumento mostrando que o tempo de execução de PARTITION sobre um subarranjo de tamanho n é $\Theta(n)$.

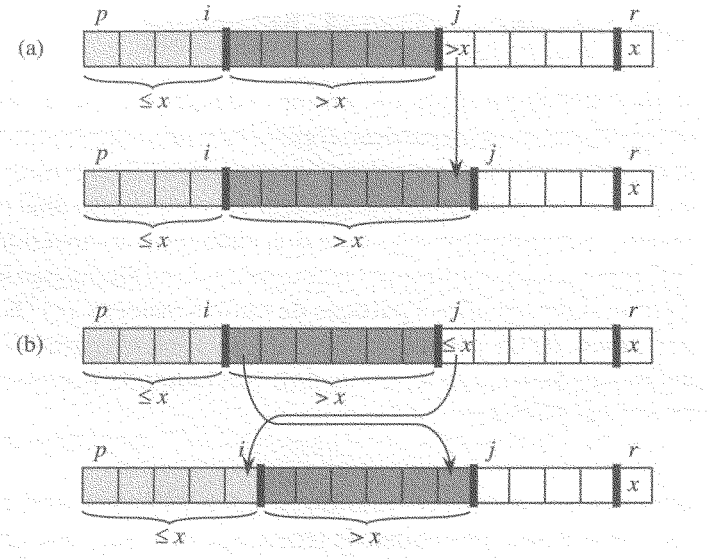

FIGURA 7.3 Os dois casos para uma iteração do procedimento PARTITION. (**a**) Se $A[j] > x$, a única ação é incrementar j, o que mantém o loop invariante. (**b**) Se $A[j] \leq x$, o índice i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Novamente, o loop invariante é mantido

7.1-4

De que maneira você modificaria QUICKSORT para fazer a ordenação em ordem não crescente?

7.2 O desempenho de quicksort

O tempo de execução de quicksort depende do fato de o particionamento ser balanceado ou não balanceado, e isso por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção. Nesta seção, investigaremos informalmente como o quicksort é executado sob as hipóteses de particionamento balanceado e particionamento não balanceado.

Particionamento no pior caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com $n - 1$ elementos e um com 0 elementos. (Essa afirmativa é demonstrada na Seção 7.4.1.) Vamos supor que esse particionamento não balanceado surge em cada chamada recursiva. O particionamento custa o tempo $\Theta(n)$. Tendo em vista que a chamada recursiva sobre um arranjo de tamanho 0 simplesmente retorna, $T(0) = \Theta(1)$, e a recorrência para o tempo de execução é

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Intuitivamente, se somarmos os custos incorridos a cada nível da recursão, conseguimos uma série aritmética (equação (A.2)), que tem o valor $\Theta(n^2)$. Na realidade, é simples usar o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$. (Veja o Exercício 7.2-1.)

Portanto, se o particionamento é não balanceado de modo máximo em cada nível recursivo do algoritmo, o tempo de execução é $\Theta(n^2)$. Por conseguinte, o tempo de execução do pior caso do quicksort não é melhor que o da ordenação por inserção. Além disso, o tempo de execução $\Theta(n^2)$ ocorre quando o arranjo de entrada já está completamente ordenado – uma situação comum na qual a ordenação por inserção é executada no tempo $O(n)$.

Particionamento do melhor caso

Na divisão mais uniforme possível, PARTITION produz dois subproblemas, cada um de tamanho não maior que $n/2$, pois um tem tamanho $\lfloor n/2 \rfloor$ e o outro tem tamanho $\lceil n/2 \rceil - 1$. Nesse caso, o quicksort é executado com muito maior rapidez. A recorrência pelo tempo de execução é então

$$T(n) \leq 2T(n/2) + \Theta(n)$$

que, pelo caso 2 do teorema mestre (Teorema 4.1) tem a solução $T(n) = O(n \lg n)$. Desse modo, o balanceamento equilibrado dos dois lados da partição em cada nível da recursão produz um algoritmo assintoticamente mais rápido.

Particionamento balanceado

O tempo de execução do caso médio de quicksort é muito mais próximo do melhor caso que do pior caso, como mostrarão as análises da Seção 7.4. A chave para compreender por que isso poderia ser verdade é entender como o equilíbrio do particionamento se reflete na recorrência que descreve o tempo de execução.

Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que a princípio parece bastante desequilibrada. Então, obtemos a recorrência

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

no tempo de execução de quicksort, onde incluímos explicitamente a constante c oculta no termo $\Theta(n)$. A Figura 7.4 mostra a árvore de recursão para essa recorrência. Note que todo nível da árvore tem custo cn, até ser alcançada uma condição limite à profundidade $\log_{10} n = \Theta(\lg n)$, e então os níveis têm o custo máximo cn. A recursão termina na profundidade $\log_{10/9} n = \Theta(\lg n)$. O custo total do quicksort é portanto $O(n \lg n)$. Desse modo, com uma divisão na proporção de

9 para 1 em todo nível de recursão, o que intuitivamente parece bastante desequilibrado, o quicksort é executado no tempo $O(n \lg n)$ – assintoticamente o mesmo tempo que teríamos se a divisão fosse feita exatamente ao meio. De fato, até mesmo uma divisão de 99 para 1 produz um tempo de execução igual a $O(n \lg n)$. A razão é que qualquer divisão de proporcionalidade *constante* produz uma árvore de recursão de profundidade $\Theta(\lg n)$, onde o custo em cada nível é $O(n)$. Portanto, o tempo de execução será então $O(n \lg n)$ sempre que a divisão tiver proporcionalidade constante.

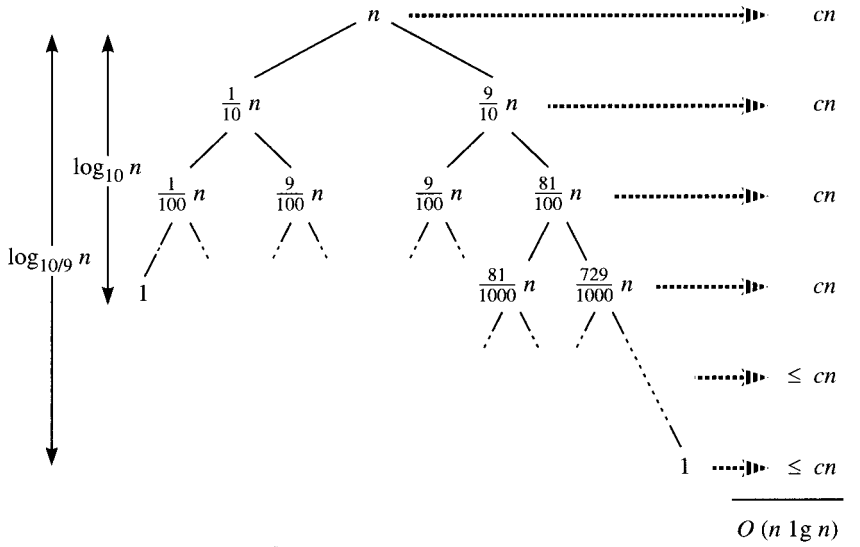

FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo $\Theta(n)$

Intuição para o caso médio

Para desenvolver uma noção clara do caso médio de quicksort, devemos fazer uma suposição sobre a freqüência com que esperamos encontrar as várias entradas. O comportamento de quicksort é determinado pela ordenação relativa dos valores nos elementos do arranjo dados como entrada, e não pelos valores específicos no arranjo. Como em nossa análise probabilística do problema da contratação na Seção 5.2, iremos supor por enquanto que todas as permutações dos números de entrada são igualmente prováveis.

Quando executamos o quicksort sobre um arranjo de entrada aleatório, é improvável que o particionamento sempre ocorra do mesmo modo em todo nível, como nossa análise informal pressupôs. Esperamos que algumas divisões sejam razoavelmente bem equilibradas e que algumas sejam bastante desequilibradas. Por exemplo, o Exercício 7.2-6 lhe pede para mostrar que, em cerca de 80% do tempo, PARTITION produz uma divisão mais equilibrada que 9 para 1, e mais ou menos em 20% do tempo ele produz uma divisão menos equilibrada que 9 para 1.

No caso médio, PARTITION produz uma mistura de divisões "boas" e "ruins". Em uma árvore de recursão para uma execução do caso médio de PARTITION, as divisões boas e ruins estão distribuídas aleatoriamente ao longo da árvore. Porém, suponha para fins de intuição, que as divisões boas e ruins alternem seus níveis na árvore, e que as divisões boas sejam divisões do melhor caso e as divisões ruins sejam divisões do pior caso. A Figura 7.5(a) mostra as divisões em dois níveis consecutivos na árvore de recursão. Na raiz da árvore, o custo é n para particionamento e os subarranjos produzidos têm tamanhos $n - 1$ e 1: o pior caso. No nível seguinte, o subarranjo de tamanho $n - 1$ é particionado no melhor caso em dois subarranjos de tamanho $(n - 1)/2 - 1$ e $(n - 1)/2$. Vamos supor que o custo da condição limite seja 1 para o subarranjo de tamanho 0.

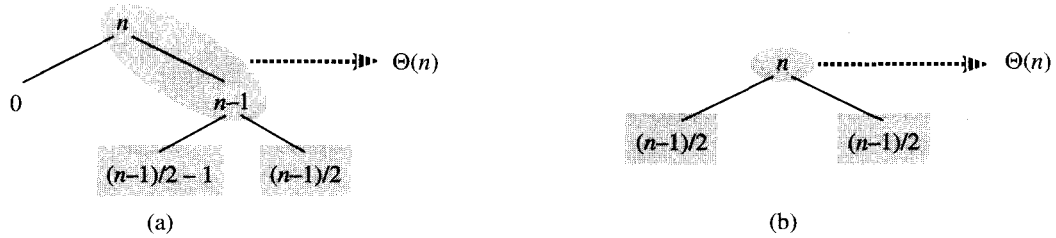

FIGURA 7.5 **(a)** Dois níveis de uma árvore de recursão para quicksort. O particionamento na raiz custa n e produz uma divisão "ruim": dois subarranjos de tamanhos 0 e $n-1$. O particionamento do subarranjo de tamanho $n-1$ custa $n-1$ e produz uma divisão "boa": subarranjos de tamanho $(n-1)/2-1$ e $(n-1)/2$. **(b)** Um único nível de uma árvore de recursão que está muito bem equilibrada. Em ambas as partes, o custo de particionamento para os subproblemas mostrados com sombreamento elíptico é $\Theta(n)$. Ainda assim, os subproblemas que restam para serem resolvidos em (a), mostrados com sombreamento retangular, não são maiores que os subproblemas correspondentes que restam para serem resolvidos em (b)

A combinação da divisão ruim seguida pela divisão boa produz três subarranjos de tamanhos 0, $(n-1)/2-1$ e $(n-1)/2$, a um custo de particionamento combinado de $\Theta(n) + \Theta(n-1) = \Theta(n)$. Certamente, essa situação não é pior que a da Figura 7.5(b), ou seja, um único nível de particionamento que produz dois subarranjos de tamanho $(n-1)/2$, ao custo $\Theta(n)$. Ainda assim, essa última situação é equilibrada! Intuitivamente, o custo $\Theta(n-1)$ da divisão ruim pode ser absorvido no custo $\Theta(n)$ da divisão boa, e a divisão resultante é boa. Desse modo, o tempo de execução do quicksort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao tempo de execução para divisões boas sozinhas: ainda $O(n \lg n)$, mas com uma constante ligeiramente maior oculta pela notação de O. Faremos uma análise rigorosa do caso médio na Seção 7.4.2.

Exercícios

7.2-1
Use o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$, como afirmamos no início da Seção 7.2.

7.2-2
Qual é o tempo de execução de QUICKSORT quando todos os elementos do arranjo A têm o mesmo valor?

7.2-3
Mostre que o tempo de execução de QUICKSORT é $\Theta(n^2)$ quando o arranjo A contém elementos distintos e está ordenado em ordem decrescente.

7.2-4
Os bancos freqüentemente registram transações em uma conta na ordem dos horários das transações, mas muitas pessoas gostam de receber seus extratos bancários com os cheques relacionados na ordem de número do cheque. Em geral, as pessoas preenchem seus cheques na ordem do número do cheque, e os comerciantes normalmente os descontam com uma presteza razoável. Portanto, o problema de converter a ordenação pela hora da transação na ordenação pelo número do cheque é o problema de ordenar uma entrada quase ordenada. Demonstre que o procedimento INSERTION-SORT tenderia a superar o procedimento QUICKSORT nesse problema.

7.2-5
Suponha que as divisões em todo nível de quicksort estejam na proporção $1-\alpha$ para α, onde $0 < \alpha \leq 1/2$ é uma constante. Mostre que a profundidade mínima de uma folha na árvore de recursão é aproximadamente $-\lg n/\lg \alpha$ e a profundidade máxima é aproximadamente $-\lg n/\lg(1-\alpha)$. (Não se preocupe com o arredondamento até inteiro.)

7.2-6 ⋆

Demonstre que, para qualquer constante $0 < \alpha \leq 1/2$, a probabilidade de que, em um arranjo de entradas aleatórias, PARTITION produza uma divisão mais equilibrada que $1 - \alpha$ para α é aproximadamente $1 - 2\alpha$.

7.3 Uma versão aleatória de quicksort

Na exploração do comportamento do caso médio de quicksort, fizemos uma suposição de que todas as permutações dos números de entrada são igualmente prováveis. Porém, em uma situação de engenharia nem sempre podemos esperar que ela se mantenha válida. (Ver Exercício 7.2-4.) Como vimos na Seção 5.3, às vezes adicionamos um caráter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas. Muitas pessoas consideram a versão aleatória resultante de quicksort o algoritmo de ordenação preferido para entrada grandes o suficiente.

Na Seção 5.3, tornamos nosso algoritmo aleatório permutando explicitamente a entrada. Também poderíamos fazer isso para quicksort, mas uma técnica de aleatoriedade diferente, chamada amostragem aleatória, produz uma análise mais simples. Em vez de sempre usar $A[r]$ como pivô, usaremos um elemento escolhido ao acaso a partir do subarranjo $A[p \,..\, r]$. Faremos isso permutando o elemento $A[r]$ com um elemento escolhido ao acaso de $A[p \,..\, r]$. Essa modificação, em que fazemos a amostragem aleatória do intervalo $p, \ldots, r$, assegura que o elemento pivô $x = A[r]$ tem a mesma probabilidade de ser qualquer um dos $r - p + 1$ elementos do subarranjo. Como o elemento pivô é escolhido ao acaso, esperamos que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média.

As mudanças em PARTITION e QUICKSORT são pequenas. No novo procedimento de partição, simplesmente implementamos a troca antes do particionamento real:

```
RANDOMIZED-PARTITION(A, p, r)
1 i ← RANDOM(p, r)
2 trocar A[p] ↔ A[i]
3 return PARTITION(A, p, r)
```

O novo quicksort chama RANDOMIZED-PARTITION em lugar de PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1 if p < r
2   then q ← RANDOMIZED-PARTITION(A, p, r)
3        RANDOMIZED-QUICKSORT(A, p, q − 1)
4        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Analisaremos esse algoritmo na próxima seção.

Exercícios

7.3-1

Por que analisamos o desempenho do caso médio de um algoritmo aleatório e não seu desempenho no pior caso?

7.3-2

Durante a execução do procedimento RANDOMIZED-QUICKSORT, quantas chamadas são feitas ao gerador de números aleatórios RANDOM no pior caso? E no melhor caso? Dê a resposta em termos de notação Θ.

7.4 Análise de quicksort

A Seção 7.2 forneceu alguma intuição sobre o comportamento do pior caso do quicksort e sobre o motivo pelo qual esperamos que ele funcione rapidamente. Nesta seção, analisaremos o comportamento do quicksort de forma mais rigorosa. Começaremos com uma análise do pior caso, que se aplica a QUICKSORT ou a RANDOMIZED-QUICKSORT, e concluímos com uma análise do caso médio de RANDOMIZED-QUICKSORT.

7.4.1 Análise do pior caso

Vimos na Seção 7.2 que uma divisão do pior caso em todo nível de recursão do quicksort produz um tempo de execução igual a $\Theta(n^2)$ que, intuitivamente, é o tempo de execução do pior caso do algoritmo. Agora, vamos provar essa afirmação.

Usando o método de substituição (ver Seção 4.1), podemos mostrar que o tempo de execução de quicksort é $O(n^2)$. Seja $T(n)$ o tempo no pior caso para o procedimento QUICKSORT sobre uma entrada de tamanho n. Então, temos a recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

onde o parâmetro q varia de 0 a $n-1$, porque o procedimento PARTITION produz dois subproblemas com tamanho total $n-1$. Supomos que $T(n) \le cn^2$ para alguma constante c. Pela substituição dessa suposição na recorrência (7.1), obtemos

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

A expressão $q^2 + (n-q-1)^2$ alcança um máximo sobre o intervalo $0 \le q \le n-1$ do parâmetro em um dos pontos extremos, como pode ser visto pelo fato da segunda derivada da expressão com relação a q ser positiva (ver Exercício 7.4-3). Essa observação nos dá o limite $\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuando com nossa definição do limite de $T(n)$, obtemos

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

pois podemos escolher a constante c grande o suficiente para que o termo $c(2n-1)$ domine o termo $\Theta(n)$. Portanto, $T(n) = O(n^2)$. Vimos na Seção 7.2 um caso específico em que quicksort demora o tempo $\Omega(n^2)$: quando o particionamento é desequilibrado. Como alternativa, o Exercício 7.4-1 lhe pede para mostrar que a recorrência (7.1) tem uma solução $T(n) = \Omega(n^2)$. Desse modo, o tempo de execução (no pior caso) de quicksort é $\Theta(n^2)$.

7.4.2 Tempo de execução esperado

Já fornecemos um argumento intuitivo sobre o motivo pelo qual o tempo de execução do caso médio de RANDOMIZED-QUICKSORT é $O(n \lg n)$: se, em cada nível de recursão, a divisão induzida por RANDOMIZED-PARTITION colocar qualquer fração constante dos elementos em um lado da partição, então a árvore de recursão terá a profundidade $\Theta(\lg n)$, e o trabalho $O(n)$ será

executado em cada nível. Ainda que sejam adicionados novos níveis com a divisão mais desequilibrada possível entre esses níveis, o tempo total continuará a ser $O(n \lg n)$. Podemos analisar o tempo de execução esperado de RANDOMIZED-QUICKSORT com precisão, compreendendo primeiro como o procedimento de particionamento opera, e depois usando essa compreensão para derivar um limite $O(n \lg n)$ sobre o tempo de execução esperado. Esse limite superior no tempo de execução esperado, combinado com o limite no melhor caso $\Theta(n \lg n)$ que vimos na Seção 7.2, resulta em um tempo de execução esperado $\Theta(n \lg n)$.

Tempo de execução e comparações

O tempo de execução de QUICKSORT é dominado pelo tempo gasto no procedimento PARTITION. Toda vez que o procedimento PARTITION é chamado, um elemento pivô é selecionado, e esse elemento nunca é incluído em quaisquer chamadas recursivas futuras a QUICKSORT e PARTITION. Desse modo, pode haver no máximo n chamadas a PARTITION durante a execução inteira do algoritmo de quicksort. Uma chamada a PARTITION demora o tempo $O(1)$ mais um período de tempo proporcional ao número de iterações do loop **for** das linhas 3 a 6. Cada iteração desse loop **for** executa uma comparação na linha 4, comparando o elemento pivô a outro elemento do arranjo A. Assim, se pudermos contar o número total de vezes que a linha 4 é executada, poderemos limitar o tempo total gasto no loop **for** durante toda a execução de QUICKSORT.

Lema 7.1
Seja X o número de comparações executadas na linha 4 de PARTITION por toda a execução de QUICKSORT sobre um arranjo de n elementos. Então, o tempo de execução de QUICKSORT é $O(n + X)$.

Prova Pela discussão anterior, existem n chamadas a PARTITION, cada uma das quais faz uma proporção constante do trabalho e depois executa o loop **for** um certo número de vezes. Cada iteração do loop **for** executa a linha 4. ■

Portanto, nossa meta é calcular X, o número total de comparações executadas em todas as chamadas a PARTITION. Não tentaremos analisar quantas comparações são feitas em *cada* chamada a PARTITION. Em vez disso, derivaremos um limite global sobre o número total de comparações. Para fazê-lo, devemos reconhecer quando o algoritmo compara dois elementos do arranjo e quando ele não o faz. Para facilitar a análise, renomeamos os elementos do arranjo A como $z_1, z_2, \ldots, z_n$, com z_i sendo o i-ésimo menor elemento. Também definimos o conjunto $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ como o conjunto de elementos entre z_i e z_j, inclusive.

Quando o algoritmo compara z_i e z_j? Para responder a essa pergunta, primeiro observamos que cada par de elementos é comparado no máximo uma vez. Por quê? Os elementos são comparados apenas ao elemento pivô e, depois que uma chamada específica de PARTITION termina, o elemento pivô usado nessa chamada nunca é comparado novamente a quaisquer outros elementos.

Nossa análise utiliza variáveis indicadoras aleatórias (consulte a Seção 5.2). Definimos

$$X_{ij} = \mathrm{I}\{z_i \text{ é comparado a } z_j\},$$

onde estamos considerando se a comparação tem lugar em qualquer instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chamada de PARTITION. Tendo em vista que cada par é comparado no máximo uma vez, podemos caracterizar facilmente o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Tomando as expectativas em ambos os lados, e depois usando a linearidade de expectativa e o Lema 5.1, obtemos

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ é comparado a } z_j\} . \end{aligned} \tag{7.2}$$

Resta calcular Pr $\{z_i$ é comparado a $z_j\}$. Nossa análise parte do princípio de que cada pivô é escolhido ao acaso e de forma independente.

É útil imaginar quando dois itens *não* são comparados. Considere uma entrada para quicksort dos números 1 a 10 (em qualquer ordem) e suponha que o primeiro elemento pivô seja 7. Então, a primeira chamada a PARTITION separa os números em dois conjuntos: {1, 2, 3, 4, 5, 6} e {8, 9, 10}. Fazendo-se isso, o elemento pivô 7 é comparado a todos os outros elementos, mas nenhum número do primeiro conjunto (por exemplo, 2) é ou jamais será comparado a qualquer número do segundo conjunto (por exemplo, 9).

Em geral, uma vez que um pivô x é escolhido com $z_i < x < z_j$, sabemos que z_i e z_j não podem ser comparados em qualquer momento subseqüente. Se, por outro lado, z_i for escolhido como um pivô antes de qualquer outro item em Z_{ij}, então z_i será comparado a cada item em Z_{ij}, exceto ele mesmo. De modo semelhante, se z_j for escolhido como pivô antes de qualquer outro item em Z_{ij}, então z_j será comparado a cada item em Z_{ij}, exceto ele próprio. Em nosso exemplo, os valores 7 e 9 são comparados porque 7 é o primeiro item de $Z_{7,9}$ a ser escolhido como pivô. Em contraste, 2 e 9 nunca serão comparados, porque o primeiro elemento pivô escolhido de $Z_{2,9}$ é 7. Desse modo, z_i e z_j são comparados se e somente se o primeiro elemento a ser escolhido como pivô de Z_{ij} é z_i ou z_j.

Agora, calculamos a probabilidade de que esse evento ocorra. Antes do ponto em que um elemento de Z_{ij} é escolhido como pivô, todo o conjunto Z_{ij} está reunido na mesma partição. Por conseguinte, qualquer elemento de Z_{ij} tem igual probabilidade de ser o primeiro escolhido como pivô. Pelo fato do conjunto Z_{ij} ter $j - i + 1$ elementos e tendo em vista que os pivôs são escolhidos ao acaso e de forma independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como pivô é $1/(j - i + 1)$. Desse modo, temos

$$\begin{aligned} \Pr\{z_i \text{ é comparado a } z_j\} &= \Pr\{z_i \text{ ou } z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \Pr\{z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &\quad + \Pr\{z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} . \end{aligned} \tag{7.3}$$

A segunda linha se segue porque os dois eventos são mutuamente exclusivos. Combinando as equações (7.2) e (7.3), obtemos

$$\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

Podemos avaliar essa soma usando uma troca de variáveis ($k = j - i$) e o limite sobre a série harmônica na equação (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}$$

Desse modo concluímos que, usando-se RANDOMIZED-PARTITION, o tempo de execução esperado de quicksort é $O(n \lg n)$.

Exercícios

7.4-1
Mostre que, na recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4-2
Mostre que o tempo de execução do quicksort no melhor caso é $\Omega(n \lg n)$.

7.4-3
Mostre que $q^2 + (n-q-1)^2$ alcança um máximo sobre $q = 0, 1, \ldots, n-1$ quando $q = 0$ ou $q = n-1$.

7.4-4
Mostre que o tempo de execução esperado do procedimento RANDOMIZED-QUICKSORT é $\Omega(n \lg n)$.

7.4-5
O tempo de execução do quicksort pode ser melhorado na prática, aproveitando-se o tempo de execução muito pequeno da ordenação por inserção quando sua entrada se encontra "quase" ordenada. Quando o quicksort for chamado em um subarranjo com menos de k elementos, deixe-o simplesmente retornar sem ordenar o subarranjo. Após o retorno da chamada de alto nível a quicksort, execute a ordenação por inserção sobre o arranjo inteiro, a fim de concluir o processo de ordenação. Mostre que esse algoritmo de ordenação é executado no tempo esperado $O(nk + n \lg(n/k))$. Como k deve ser escolhido, tanto na teoria quanto na prática?

7.4-6 ★
Considere a modificação do procedimento PARTITION pela escolha aleatória de três elementos do arranjo A e pelo particionamento sobre sua mediana (o valor médio dos três elementos). Faça a aproximação da probabilidade de se obter na pior das hipóteses uma divisão de α para $(1 - \alpha)$, como uma função de α no intervalo $0 < \alpha < 1$.

Problemas

7-1 Correção da partição de Hoare

A versão de PARTITION dada neste capítulo não é o algoritmo de particionamento original. Aqui está o algoritmo de partição original, devido a T. Hoare:

```
HOARE-PARTITION(A, p, r)
 1 x ← A[p]
 2 i ← p − 1
 3 j ← r + 1
 4 while TRUE
 5     do repeat j ← j − 1
 6          until A[j] ≤ x
 7        repeat i ← i + 1
 8          until A[i] ≥ x
 9        if i < j
10          then trocar A[i] ↔ A[j]
11          else return j
```

a. Demonstre a operação de HOARE-PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$, mostrando os valores do arranjo e os valores auxiliares depois de cada iteração do loop **for** das linhas 4 a 11.

As três perguntas seguintes lhe pedem para apresentar um argumento cuidadoso de que o procedimento HOARE-PARTITION é correto. Prove que:

b. Os índices i e j são tais que nunca acessamos um elemento de A fora do subarranjo $A[p..r]$.

c. Quando HOARE-PARTITION termina, ele retorna um valor j tal que $p \leq j < r$.

d. Todo elemento de $A[p..j]$ é menor que ou igual a todo elemento de $A[j+1..r]$ quando HOARE-PARTITION termina.

O procedimento PARTITION da Seção 7.1 separa o valor do pivô (originalmente em $A[r]$) das duas partições que ele forma. Por outro lado, o procedimento HOARE-PARTITION sempre insere o valor do pivô (originalmente em $A[p]$) em uma das duas partições $A[p..j]$ e $A[j+1..r]$. Como $p \leq j < r$, essa divisão é sempre não trivial.

e. Reescreva o procedimento QUICKSORT para usar HOARE-PARTITION.

7-2 Análise alternativa de quicksort

Uma análise alternativa do tempo de execução de quicksort aleatório se concentra no tempo de execução esperado de cada chamada recursiva individual a QUICKSORT, em vez de se ocupar do número de comparações executadas.

a. Demonstre que, dado um arranjo de tamanho n, a probabilidade de que qualquer elemento específico seja escolhido como pivô é $1/n$. Use isso para definir variáveis indicadoras aleatórias $X_i = \mathrm{I}\{i\text{-ésimo menor elemento é escolhido como pivô}\}$. Qual é $\mathrm{E}[X_i]$?

b. Seja $T(n)$ uma variável aleatória denotando o tempo de execução de quicksort sobre um arranjo de tamanho n. Demonstre que

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \tag{7.5}$$

c. Mostre que a equação (7.5) é simplificada para

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1}\mathrm{E}[T(q)] + \Theta(n) . \tag{7.6}$$

d. Mostre que

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \tag{7.7}$$

(*Sugestão*: Divida o somatório em duas partes, uma para $k = 1, 2, \ldots, \lceil n/2 \rceil - 1$ e uma para $k = \lceil n/2 \rceil, \ldots, n - 1$.)

e. Usando o limite da equação (7.7), mostre que a recorrência na equação (7.6) tem a solução $\mathrm{E}[T(n)] = \Theta(n \lg n)$. (*Sugestão*: Mostre, por substituição, que $\mathrm{E}[T(n)] \leq an \log n - bn$ para algumas constantes positivas a e b.)

7-3 Ordenação do pateta

Os professores Howard, Fine e Howard propuseram o seguinte algoritmo de ordenação "elegante":

```
STOOGE-SORT(A, i, j)
1 if A[i] > A[j]
2    then trocar A[i] ↔ A[j]
3 if i + 1 ≥ j
4    then return
5 k ← ⌊(j − i + 1)/3⌋          ▷ Arredonda para menos.
6 STOOGE-SORT(A, i, j − k)     ▷ Primeiros dois terços.
7 STOOGE-SORT(A, i + k, j)     ▷ Últimos dois terços.
8 STOOGE-SORT(A, i, j − k)     ▷ Primeiros dois terços novamente.
```

a. Mostre que, se n = *comprimento*$[A]$, então STOOGE-SORT(A, 1, *comprimento*$[A]$) ordena corretamente o arranjo de entrada $A[1 .. n]$.

b. Forneça uma recorrência para o tempo de execução no pior caso de STOOGE-SORT e um limite assintótico restrito (notação Θ) sobre o tempo de execução no pior caso.

c. Compare o tempo de execução no pior caso de STOOGE-SORT com o da ordenação por inserção, da ordenação por intercalação, de heapsort e de quicksort. Os professores merecem a estabilidade no emprego?

7-4 Profundidade de pilha para quicksort

O algoritmo QUICKSORT da Seção 7.1 contém duas chamadas recursivas a ele próprio. Após a chamada a PARTITION, o subarranjo da esquerda é ordenado recursivamente, e depois o subarranjo da direita é ordenado recursivamente. A segunda chamada recursiva em QUICKSORT não é realmente necessária; ela pode ser evitada pelo uso de uma estrutura de controle iterativa. Essa técnica, chamada ***recursão do final***, é automaticamente fornecida por bons compiladores. Considere a versão de quicksort a seguir, que simula a recursão do final.

```
QUICKSORT'(A, p, r)
1 while p < r
2    do ▷ Particiona e ordena o subarranjo esquerdo
3       q ← PARTITION(A, p, r)
4       QUICKSORT'(A, p, q − 1)
5       p ← q + 1
```

a. Mostre que QUICKSORT'(A, 1, *comprimento*[A]) ordena corretamente o arranjo A.

Os compiladores normalmente executam procedimentos recursivos usando uma ***pilha*** que contém informações pertinentes, inclusive os valores de parâmetros, para cada chamada recursiva. As informações para a chamada mais recente estão na parte superior da pilha, e as informações para a chamada inicial encontram-se na parte inferior. Quando um procedimento é invocado, suas informações são ***empurradas*** sobre a pilha; quando ele termina, suas informações são ***extraídas***. Tendo em vista nossa suposição de que os parâmetros de arranjos são na realidade representados por ponteiros, as informações correspondentes a cada chamada de procedimento na pilha exigem o espaço de pilha $O(1)$. A ***profundidade de pilha*** é a quantidade máxima de espaço da pilha usado em qualquer instante durante uma computação.

b. Descreva um cenário no qual a profundidade de pilha de QUICKSORT' é $\Theta(n)$ sobre um arranjo de entrada de n elementos.

c. Modifique o código de QUICKSORT' de tal modo que a profundidade de pilha no pior caso seja $\Theta(\lg n)$. Mantenha o tempo de execução esperado $O(n \lg n)$ do algoritmo.

7-5 Partição de mediana de 3

Um modo de melhorar o procedimento RANDOMIZED-QUICKSORT é criar uma partição em torno de um elemento x escolhido com maior cuidado que a simples escolha de um elemento aleatório do subarranjo. Uma abordagem comum é o método da ***mediana de 3***: escolha x como a mediana (o elemento intermediário) de um conjunto de 3 elementos selecionados aleatoriamente a partir do subarranjo. Para esse problema, vamos supor que os elementos no arranjo de entrada $A[1 .. n]$ sejam distintos e que $n \geq 3$. Denotamos o arranjo de saída ordenado por $A'[1.. n]$. Usando o método da mediana de 3 para escolher o elemento pivô x, defina $p_i = \Pr\{x = A'[i]\}$.

a. Dê uma fórmula exata para p_i como uma função de n e i para $i = 2, 3, ..., n - 1$. (Observe que $p_1 = p_n = 0$.)

b. Por qual valor aumentamos a probabilidade de escolher $x = A'[\lfloor (n + 1)/2 \rfloor]$, a mediana de $A[1.. n]$, em comparação com a implementação comum? Suponha que $n \to \infty$ e forneça a razão de limitação dessas probabilidades.

c. Se definimos uma "boa" divisão com o significado de escolher o pivô como $x = A'[i]$, onde $n/3 \leq i \leq 2n/3$, por qual quantidade aumentamos a probabilidade de se obter uma boa divisão em comparação com a implementação comum? (*Sugestão*: Faça uma aproximação da soma por uma integral.)

d. Mostre que, no tempo de execução $\Omega(n \lg n)$ de quicksort, o método da mediana de 3 só afeta o fator constante.

7-6 Ordenação nebulosa de intervalos

Considere um problema de ordenação no qual os números não são conhecidos exatamente. Em vez disso, para cada número, conhecemos um intervalo sobre a linha real a que ele pertence. Ou seja, temos n intervalos fechados da forma $[a_i, b_i]$, onde $a_i \leq b_i$. O objetivo é fazer a ***ordenação nebulosa*** desses intervalos, isto é, produzir uma permutação $\langle i_1, i_2, ..., i_n \rangle$ dos intervalos tal que exista $c_j \in [a_{i_j}, b_{i_j}]$ que satisfaça a $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Projete um algoritmo para ordenação do pateta de n intervalos. Seu algoritmo devia ter a estrutura geral de um algoritmo que faz o quicksort das extremidades esquerdas (os valores a_i), mas deve tirar proveito da sobreposição de intervalos para melhorar o tempo de execução. (À medida que os intervalos se sobrepõem cada vez mais, o problema de fazer a ordenação do pateta dos intervalos fica cada vez mais fácil. Seu algoritmo deve aproveitar tal sobreposição, desde que ela exista.)

b. Demonstre que seu algoritmo é executado no tempo esperado $\Theta(n \lg n)$ em geral, mas funciona no tempo esperado $\Theta(n)$ quando todos os intervalos se sobrepõem (isto é, quando existe um valor x tal que $x \in [a_i, b_i]$ para todo i). O algoritmo não deve verificar esse caso de forma explícita; em vez disso, seu desempenho deve melhorar naturalmente à medida que aumentar a proporção de sobreposição.

Notas do capítulo

O procedimento quicksort foi criado por Hoare [147]; a versão de Hoare aparece no Problema 7-1. O procedimento PARTITION dado na Seção 7.1 se deve a N. Lomuto. A análise da Seção 7.4 se deve a Avrim Blum. Sedgewick [268] e Bentley [40] fornecem uma boa referência sobre os detalhes de implementação e como eles são importantes.

McIlroy [216] mostrou como engenheiro um "adversário matador" que produz um arranjo sobre o qual virtualmente qualquer implementação de quicksort demora o tempo $\Theta(n^2)$. Se a implementação é aleatória, o adversário produz o arranjo depois de ver as escolhas ao acaso do algoritmo de quicksort.

Capítulo 8

Ordenação em tempo linear

Apresentamos até agora diversos algoritmos que podem ordenar n números no tempo $O(n \lg n)$. A ordenação por intercalação e o heapsort alcançam esse limite superior no pior caso; quicksort o alcança na média. Além disso, para cada um desses algoritmos, podemos produzir uma seqüência de n números de entrada que faz o algoritmo ser executado no tempo $\Omega(n \lg n)$.

Esses algoritmos compartilham uma propriedade interessante: *a seqüência ordenada que eles determinam* se *baseia apenas em comparações entre os elementos de entrada*. Chamamos esses algoritmos de ordenação de ***ordenações por comparação***. Todos os algoritmos de ordenação apresentados até agora são portanto ordenações por comparação.

Na Seção 8.1, provaremos que qualquer ordenação por comparação deve efetuar $\Omega(n \lg n)$ comparações no pior caso para ordenar n elementos. Desse modo, a ordenação por intercalação e heapsort são assintoticamente ótimas, e não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

As Seções 8.2, 8.3 e 8.4 examinam três algoritmos de ordenação – ordenação por contagem, radix sort (ordenação da raiz) e bucket sort (ordenação por balde) – que são executados em tempo linear. É desnecessário dizer que esses algoritmos utilizam outras operações diferentes de comparações para determinar a seqüência ordenada. Em conseqüência disso, o limite inferior $\Omega(n \lg n)$ não se aplica a eles.

8.1 Limites inferiores para ordenação

Em uma ordenação por comparação, usamos apenas comparações entre elementos para obter informações de ordem sobre uma seqüência de entrada $\langle a_1, a_2, \ldots, a_n \rangle$. Ou seja, dados dois elementos a_i e a_j, executamos um dos testes $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ ou $a_i > a_j$, para determinar sua ordem relativa. Podemos inspecionar os valores dos elementos ou obter informações de ordem sobre eles de qualquer outro modo.

Nesta seção, vamos supor sem perda de generalidade que todos os elementos de entrada são distintos. Dada essa hipótese, as comparações da forma $a_i = a_j$ são inúteis; assim, podemos supor que não é feita nenhuma comparação dessa forma. Também observamos que as comparações $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ são todas equivalentes, em virtude de produzirem informações idênticas sobre a ordem relativa de a_i e a_j. Por essa razão, vamos supor que todas as comparações têm a forma $a_i \leq a_j$.